Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 57(3), 836–845(2016)

Ognjen Orel, Slaven Zakošek, Mirta Baranović

Property Oriented Relational-To-Graph Database Conversion

DOI 10.7305/automatika.2017.02.1581
UDK 004.652.8.057.6.021:004.652.4.057.6; 004.82

Original scientific paper

Analysis of data stored in a graph enables the discovery of certain information that could be hard to see if the data were stored using some other model (e.g. relational). However, the vast majority of data in information systems today is stored in relational databases, which dominate the data management field over the last decades. In spite of the rise of NoSQL technologies, the development of new information systems is still mostly based on relational databases. Given the increasing awareness about the benefits of data analysis as well as current research interest in graph mining techniques, we aim to enable the usage of those techniques on relational data. In that regard, we propose a universal relational-to-graph data conversion algorithm which can be used in preparation of data to perform a graph mining analysis. Our approach leverages the property graph model which is mainly used by the graph databases, while maintaining the level of relational data clarity.

Key words: graph database, GDBMS, graph mining, relational-to-graph, property graph model

Konverzija relacijskih u grafovske baze podataka orijentirana na svojstva. Analiza podataka u formatu grafa omogućava pronalazak određenih informacija koje može biti vrlo teško vidjeti ako su podaci u nekom drugom formatu (npr. relacijskom). Ipak, velika većina podataka koji su danas dio informacijskih sustava pohranjena je upravo u relacijskim bazama podataka koje dominiraju tržištem u posljednjih nekoliko desetljeća. I dalje se razvoj novih informacijskih sustava uglavnom zasniva na relacijskim bazama podataka. Kako je sve veća svjesnost o vrijednosti analize podataka, kao i aktualni interes istraživanja u području tehnika dubinske analize grafova, naš je cilj omogućiti korištenje tih tehnika nad relacijskim podacima. U tom smislu, predlažemo univerzalni algoritam konverzije podataka iz relacijskog modela u graf, koji se može koristiti u pripremi podataka za izvođenje dubinske analize grafova. Naš pristup maksimalno iskorištava model grafa sa svojstvima koji je u širokoj uporabi u aktualnim grafovskim bazama podataka, u isto vrijeme zadržavajući razinu jasnoće relacijskih podataka.

Ključne riječi: grafovska baza podataka, GDBMS, dubinska analiza grafova, relacija-u-graf, model grafa sa svojstvima

1 INTRODUCTION

Graph databases and graph data models are subject of research for a very long time - the first graph data model is proposed in 1975 [1]. During the first period of research, a number of graph database models are proposed, as well as data access and manipulation languages. Graph database research set back until recently, when NoSQL databases and data storage philosophy, a bit different from relational databases and aiming towards a more efficient resource leverage, was introduced to a broader audience. Developers and researches are more aware that data that is commonly used in various fields can be described in more natural way using graph models: semantic web, social networks, navigation system locations, natural sciences, etc.

Increased need for big data analysis has affected graph data as well, resulting in a number of more or less successful proposed algorithms which fall into this category. Graph mining is mostly oriented towards graph pattern detection algorithms, which is most commonly related to frequent subgraph mining methods. Frequent topics in graph research are various measures (e.g. betweenness, centrality, closeness) and algorithms (e.g. shortest path, fastest path) as well.

Most of the data consisted in information systems today are stored in relational databases. The value of these data might be considerable, regarding the fact that some information systems are in production for long periods of time and consist of big amounts of data collected through verified business processes. Enabling the view on this data in form of a graph, brings the possibility of using graph mining methods over original relational data. Hence the focus of this research is design and implementation of universal algorithms for data conversion from relational databases to graph databases, while having graph mining methods,

specifically frequent subgraph mining algorithms, in mind.

The main contribution of our work is a method of converting the entire relational database to a graph database with corresponding formalized algorithms. Comparing to the similar works, the following state-of-the-art features are presented: 1. the use of both node's and edge's properties in the property graph data model, 2. independence of the data semantics and 3. conversion without data loss.

Our approach leverages the property graph model, a common data model used by graph databases, in order to optimize graph data storage while maintaining the relational data clarity at the same time. The resulting graph database provides a good data source for graph mining algorithms. Since the entities and their relations from the relational database are modeled as nodes and edges in the graph database, this approach focuses on the graph mining algorithms on entities instead on their attributes which may or may not be taken into account. Considering the attributes gives the broader and more relevant picture of the data, which is the focus of our future work. This approach may also serve as the crucial step in migrating the business from relational database to graph database in general. It may also provide a tool for exploring the usage of graph databases for storing and analysis of more complex data which can be found in various relational databases.

The paper is organized as follows. Related work is considered in Section 2. Some basic facts about graph data model and current graph databases are presented in Section 3. Definitions of relational and graph database elements, along with some assumptions of the databases' functions are given in Section 4. The conversion concept and detailed algorithms are shown in Section 5. We show the advantages of our work by comparing data models generated by our method and some other methods in Section 6. Future work is considered in Section 7.

2 RELATED WORK

There has been some extent of work in the area of enabling graph-oriented approach on the relational data. Soussi [2] suggests that relational database data could be transformed in some other model which can later be used to create a graph in a more simple way. Among other, those models are ER (Entity-Relationship), RDF (Resource Description Framework) and XML (eXtensible Markup Language). Even though several XML extensions intended to be used for graph description (such as GraphXML, GXL, GraphML) exist, efficient algorithms for converting relational database in some of these formats are yet to be proposed. Basic shortcoming of building a graph through some other model in between is the inability of detection of interactions between generated nodes without the appropriate ontology for necessary data and tables being built beforehand.

Several tools which enable graph-like feedback from relational database has been proposed as well. These tools enable the user, which is not aware of the underlying relational database structure, to pose a query in a format of search keywords that could be found somewhere in database tuples, and get the result in the format of graph [3]. Depending on the database structure, complex queries are internally generated, in which the tables are naturally joined, outer joined or joined in a union. Based on resulting tuples, graphs are composed, in which tables, tuples and connection between them could be represented in various ways. BANKS [4] models the tuples as the graph nodes which are connected by edges which represent foreign keys or transitive tables. Similar solutions are BLINKS [5], DBXplorer [6] and Discover [7].

Recently proposed GraphGen [8] and Aster6 [9] also provide the ability of generating a graph from a relational data. These tools aim to enable the view on relational data as an in-memory graph, relying on the user to define the data being converted. While they do enable graph mining techniques on relational data, our approach enables the permanent usage of the converted data which is stored in a graph database, as well as providing the user the ability to discover interesting facts about the data by mining through the entire database without the need of a priori defining which data should be put into the graph.

RDB2Graph [10] is the approach of transforming relational data to graph using conceptual graph [11]. The leaves of the graph created are values of the tuple's attributes. This algorithm does not support composite primary or foreign keys. While it may be suitable for frequent subgraph mining from the perspective of attributes, it is not as suitable for frequent subgraph mining from the perspective of entities. For instance, tuples with multiple overlapping attribute values may form frequent subgraphs which represent only a part of the tuple, hence producing a considerable amount of frequent subgraphs with a little relevance to entities and relations between them. DB2Graph algorithm [12] is based on RR-graph (relation-of-relations graph). RR-graph is graph in which nodes represent the relations (tables), node attributes represent values of the data in the tables, while edges represent foreign keys between tables. This approach is more appropriate regarding conversion durations, but not tested on bigger databases.

The closest work to ours, in the sense of converting the entire relational database to a graph database, is R2G [13]. However, while it uses properties with nodes, it disregards the possibility of using edges in a more active way of storing data comparing to our approach. It also may aggregate the unifiable tuples in the same node, which makes it less suitable for use in terms of graph mining over relational source.

In the context of binding relational databases and

graphs, some tools are proposed which exploit powerful relational database engines and their computational abilities as a graph storage, namely Vertexica [14] and Grail [15].

3 GRAPH DATABASES AND MODELS

The main characteristic of a graph database is the usage of graph database model. Angles defines this model as "a model where the data structures for the schema and/or instances are modeled as a (labeled) (directed) graph, or generalizations of the graph data structure, where data manipulation is expressed by graph-oriented operations and type constructors, and has integrity constraints appropriate for the graph structure" [16]. A number of graph models proposed before 2002 is shown in the same work.

The current graph databases are part of the NoSQL movement and philosophy. Most of them use some sort of property graph model. In general, data entities and relations among them are referred to as nodes and edges. In some cases, both nodes and edges can have properties attached to them, and edges can be directed from starting node to ending node.

There are a number of current graph databases available: AllegroGraph [17], Sparksee (formerly known as DEX) [18, 19], HyperGraphDB [20, 21], Neo4j [22], Infinite Graph [23], etc. A comparison of these and some other graph databases, considering data storage, data operation and manipulation features, graph data structures, representation of entities and relations, query facilities, integrity constraints and support for essential graph queries is given in [24]. Regarding graph data structures, all of the compared databases supported labeled nodes and edges and directed edges, while half of them supported node and edge attribution (properties), including DEX, InfiniteGraph and Neo4j.

4 RELATIONAL AND GRAPH DATABASE ELEMENTS

This section briefly presents basic relational and graph database elements which are used in this relational-to-graph conversion approach. Some basic functions regarding both relational and graph metadata which will be used in the approach are mentioned here as well.

4.1 Relational database

An *entity* is a subject that has an essence and characteristics that differ it from its surroundings.

An *attribute* A is a characteristic of an entity. It is also referred to as a column.

A *relational schema* R is a named set of attributes: $R = \{A_1, A_2, ..., A_n\}$.

A *relation* r is defined on a relational schema R as a finite set of tuples. Relation is also referred to as a table, and that term will be used henceforth.

A *tuple* t is a true statement in the context of a table and contains the values of its attributes. Tuple is also referred to as a row. The value of an attribute A in a tuple t is referred to as $t[A]$. In this context, this notation is also used to get a concatenated value of multiple attributes in a tuple.

Database shema **R** is a set of relational schemas: $\mathbf{R} = \{R_1, R_2, ..., R_n\}$. *Database instance* **r** on a database schema **R** is a set of relations $\{r_1, r_2, ..., r_n\}$ such that a table r_i is defined on a relational schema R_i.

4.2 Graph database

A graph database with labeled and property-enabled nodes and edges can be defined as:

$$\mathcal{G} = (V, E, L, \lambda, P, \pi) \tag{1}$$

where $V = \{v_1, v_2, ..., v_n\}$ is the set of nodes and $E = \{e_1, e_2, ..., e_n\}$ is the set of edges, providing each edge has two nodes it connects, $e = (v_i, v_j)$. L is the set of labels which can label a node or an edge, and $\lambda : V \cup E \rightarrow L$ is a labeling function. In the following text, labels are stated with a semi-colon in front, using a Neo4j notation, e.g. $:label$. $P = \{(K_1, V_1), (K_2, V_2), ...(K_n, V_n)\}$ is a set of properties which are key-value pairs, and $\pi : V \cup E \rightarrow P$ is a property-assigning function for nodes and edges as well.

Given the fact that there is a notion of starting and ending node of an edge, graph databases can support both directed and undirected graphs. It is most often left to the user of the database to determine which type of graph will be used. In our method, we assume the database to contain undirected graphs.

5 RELATIONAL DATABASE TO GRAPH DATABASE CONVERSION

5.1 The concept

In this section, we state the key aspects of the proposed conversion method.

1. Undirected graph which is created in the process may not necessary be connected or acyclic. Connectedness and acyclicity depends on the relational database structure and the data contained within.
2. Each node represent one tuple from a relational database table. Some edges represent a tuple, other represent the foreign key.
3. Labels are used to label different types of nodes and edges. Each node and each edge get exactly one label, even though some graph databases enable the elements to have multiple labels. Labels help determine

the type of a node or an edge and are crucial for this procedure.

4. Node's label is the name of relational database's table whose tuple is represented by that node. The same is true for the edge label if edge represents a tuple. If an edge represents the foreign key, then the name of the foreign key is used as edge's label.

5. Node's and edge's properties are key-value pairs representing tuple's attributes and their values. The attribute name becomes the key of the property, it's value becomes the value of the property. Edges representing foreign keys do not have any properties.

6. Each node or edge representing a tuple has the "id" property, which gets the value of the tuple's primary key. If the primary key is composite, then concatenated value is used. The concatenation delimiter may be chosen as the one which does not appear in data values, such as #, but the choice of concatenation delimiter is actually of no importance, because it is certain that concatenated value of all primary key attributes will be unique for a table regardless of the choice of delimiter.
   All elements (node or edges) representing tuples of a single relational database table will get the same label and unique ids, i.e. the id property is not unique throughout the entire graph, but the combination of label and id is.

7. Tables with exactly two foreign keys, whose primary key is not referenced by any other table are the relational representation of the M:N relationships among entities. Their tuples will be represented by edges. Those edges will connect two nodes which represent two referenced tuples in foreign key tables.

8. The tuples of all other tables are to be represented by nodes.

9. In the case of parallel (but not cyclic) relationships between two relational database tables (i.e. one table has two or more foreign keys referencing the same table), nodes representing those tables' tuples will have multiple edges between them, each labeled as the foreign key name. For instance, person can have place of birth and place of residence, which would be represented in a relational database as two foreign keys from $person$ table to a $place$ table. During the conversion, if the place of birth and the place of residence are the same for a specific person, then the node representing a person and the node representing that place will be connected with two different types of edges.

10. Tuple attributes which are part of the foreign key are not written in properties. That information is contained in the sole relation with the other graph elements, be it referencing node or edge. Primary key values are contained in the id properties, foreign key values are contained in the relation to another graph element, while only dependent attributes are written as properties of graph elements.

5.2 Identifying conversion metadata

Our goal is to define an efficient algorithm for exporting (unloading) the data from a relational database and importing (loading) it to a graph database. This implies each table's data should be considered a minimum number of times in the process. Therefore, before the actual conversion is done, the process should be prepared by inspecting the relational database's metadata. Relational databases in general follow Codd's relational data model rules, so database's metadata (description of tables, primary and foreign keys, etc.) is accessible in the same manner as the "normal" data is.

Through the metadata analysis, the following should be identified:

- tables whose tuples will become edges
- tables whose tuples will become nodes
- cyclic relationships formed by foreign keys, which will form cyclic loops in the graph
- order of creation nodes and edges
- attributes in each table which are part of a primary key, which will be used as ids
- attributes in each table which are part of foreign keys, which will be used in forming edges
- attributes in each table which are not part of any key, which will be used in forming node and edge properties

Algorithms performing these tasks work with database's metadata, and their performance depends on the complexity of the relational database (primarily number of tables and foreign keys), but not the size of the database itself.

For the sake of algorithm presentation, we assume the following simple functions exists in a relational database:

- $pk(r)$ which provides the set of primary key attributes of a table r,
- $fks(r)$ which provides the set of foreign keys on a table r,

- $refd(fk)$ which finds the table referenced by a foreign key fk (primary key table),
- $refing(fk)$ which finds the referencing table of a foreign key fk (foreign key table).

Also, these simple functions are presumed to be implemented in a graph database:

- $\mathcal{G}.add\_node(:label, id)$ which creates a node labeled $label$ with an id value of id,
- $\mathcal{G}.add\_edge(v_1, v_2, :\ label, [id])$ which creates an edge labeled $label$ with an optional id id between nodes v_1 and v_2,
- $\mathcal{G}.get\_node(:label, id)$ which finds and returns a node labeled $label$ with an id id,
- $x.add\_property(K, V)$ which adds a property named K with a value V to a graph element x.

5.3 Defining conversion order

The first task in defining conversion process is identification of future nodes and edges as well as the order of conversion of the tables, in order to minimize the effort of importing data. For this purpose, function $get\_migration\_order$ given in the Algorithm 1 is used. Data which is part of a table which references two other tables and is not referenced by any other table is put aside and will be converted to edges at the end of the process. These tables are listed in the *ToEdges* set.

Data which is part of a table that does not reference any other table should be converted first in a way that new nodes are created for each tuple. After that, tables which reference those tables already converted are considered. For each tuple, a new node should be made, and for each of its foreign keys, an edge connecting that node with the nodes already in database which represent referenced tuples from the primary tables. With increasing number of tables converted to graph, this process is repeated until there are no more tables which are not converted. All of these tables are put in ordered list $ToNodes$, which is used in a conversion processes.

5.4 Identifying cyclic references

Since some databases allow cyclic references among tables, those has to be taken into account while designing the migration process. Cyclic references imply that, while migrating some foreign keys to edges, there might not yet exist appropriate nodes for those edges to connect to, hence these foreign keys should be migrated after all the tuples from the referenced tables.

Algorithm 1 Defining table migration order for a database: **get_migration_order**(**r**)

Input: relational database instance **r**
Output: set of tables to migrate to edges
Output: ordered list of tables to migrate to nodes

1. $ToEdges \leftarrow \emptyset, ToNodes \leftarrow \emptyset$
2. **for all** $r \in \mathbf{r} \mid |fks(r)| = 2 \wedge r \notin \{refing(fks(s)), s \in \mathbf{r}\}$ **do**
3. $\quad ToEdges \leftarrow ToEdges \cup r$
4. **end for**
5. **repeat**
6. $\quad$ {candidates in each round are those tables that have not been selected yet and which reference only tables that are already selected}
7. $\quad candidates \leftarrow \{r \in \mathbf{r} \mid r \notin ToNodes \wedge r \notin ToEdges \wedge refing(fks(r)) \subset ToNodes\}$
8. $\quad$ **for all** $r \in candidates$ **do**
9. $\quad\quad ToNodes \leftarrow ToNodes \cup r$
10. $\quad$ **end for**
11. **until** $candidates \neq \emptyset$
12. **return** $ToEdges, ToNodes$

In general case, table r_1 can reference table r_2 via its foreign key fk_1, table r_2 can reference table r_3 via its foreign key fk_2, continuing to the table r_{n-1} referencing table r_n via foreign key fk_{n-1} and finally table r_n referencing table r_1 via foreign key fk_n thusly closing the referencing cycle. We refer to the foreign key fk_n as the cycle-closing foreign key.

The simplest form of a cyclic reference (for n = 1) is the most common - self-referencing tables are often used for representation of a hierarchical data, such as organizational units and alike.

A database table is tested for being referenced in a cyclic manner by observing its foreign keys and the tables they reference using the recursive function $find\_cycle\_ref\_for\_table$ given in the Algorithm 2.

If the referencing table of a foreign key is in fact the table from which the search started, than the cycle-closing foreign key has been found and is added to the list $CyclicFKs$ which holds all the foreign keys that form any cycle in the database instance. A list of foreign keys already visited and examined while testing a single table for being cyclically referenced is maintained to provide recursion exit. The function $find\_all\_cycle\_ref$ in the Algorithm 3 is used to ensure finding foreign keys which are part of any cycle for all the tables in a database instance.

Algorithm 2 Cyclic references detection for referenced table: **find_cycle_ref_for_table**$(referenced, fk)$

Input: $referenced$ table
Input: fk foreign key to be investigated
1. $observed \leftarrow refing(fk)$
2. **if** $observed = referenced$ **then**
3. $\quad CyclicFKs \leftarrow CyclicFKs \cup observed$
4. **end if**
5. **if** $fk \in VisitedFKs$ **then**
6. $\quad$ **return**
7. **else**
8. $\quad VisitedFKs \leftarrow VisitedFKs \cup observed$
9. **end if**
10. **for all** $fk_i \in fks(observed)$ **do**
11. $\quad find\_cycle\_ref\_for\_table(referenced, fk_i)$
12. **end for**

Algorithm 3 Cyclic references detection in a database: **find_all_cycle_ref**(**r**)

Input: relational database instance **r**
Output: set of cycle-closing foreign keys
1. $CyclicFKs \leftarrow \emptyset$
2. **for all** $r \in$ **r do**
3. $\quad VisitedFKs \leftarrow \emptyset$
4. $\quad$ **for all** $fk \in fks(r)$ **do**
5. $\quad\quad find\_cycle\_ref\_for\_table(r, fk)$
6. $\quad$ **end for**
7. **end for**
8. **return** $CyclicFKs$

5.5 Converting the data

In the end of table conversion process, data which is part of tables to be converted in edges is processed and new edges are created.

Function $table\_to\_graph$, described in Algorithm 4, provides the data migration of a single table to a graph database.

If a table is previously determined to be converted to nodes, a new node is created in the graph database for every tuple of the table. Every new node is labeled as the table's name, and is provided with an id which is a concatenated value of all the attributes which form the primary key of the table. As stated before, attributes of the table which are not part of a primary key or any foreign key are converted to node's properties, where property name is the attribute name, and property value is the attribute value in the tuple.

While processing the tuple, all foreign keys which are not part of any cyclic relationship are processed as well. For each of them, a previously created node labeled with the name of the referenced table and with id equal to concatenated values of attributes in the foreign key, is found. A new edge between newly created node and this node is created. This edge is labeled with the name of the foreign key.

Algorithm 4 Table migration: **table_to_graph**$(r, \mathcal{G})$

Input: table r with schema R
Input: graph database $\mathcal{G}$
Output: graph elements representing tuples being added to graph database
1. **for all** $t \in r$ **do**
2. $\quad id \leftarrow t[pk(r)]$
3. $\quad$ **if** $r \in ToNodes$ **then**
4. $\quad\quad v \leftarrow \mathcal{G}.add\_node(:r, id)$
5. $\quad\quad$ **for all** $A_i \in R \,|A_i \notin pk(r) \wedge A_i \notin fks(r)$ **do**
6. $\quad\quad\quad v.add\_property(A_i, t[A_i])$
7. $\quad\quad$ **end for**
8. $\quad\quad$ **for all** $fk \in fks(r) \,|fk \notin CyclicFKs$ **do**
9. $\quad\quad\quad r_{refd} \leftarrow refd(fk)$
10. $\quad\quad\quad id_{refd} \leftarrow t[fk]$
11. $\quad\quad\quad v_{refd} \leftarrow \mathcal{G}.get\_node(:r_{refd}, id_{refd})$
12. $\quad\quad\quad \mathcal{G}.add\_edge(v, v_{refd}, :fk)$
13. $\quad\quad$ **end for**
14. $\quad$ **end if**
15. $\quad$ **if** $r \in ToEdges$ **then**
16. $\quad\quad fk_1, fk_2 \leftarrow fks(r)$
17. $\quad\quad r_{refd1} \leftarrow refd(fk_1)$
18. $\quad\quad r_{refd2} \leftarrow refd(fk_2)$
19. $\quad\quad id_{refd1} \leftarrow t[fk_1]$
20. $\quad\quad id_{refd2} \leftarrow t[fk_2]$
21. $\quad\quad v_{refd1} \leftarrow \mathcal{G}.get\_node(:r_{refd1}, id_{refd1})$
22. $\quad\quad v_{refd2} \leftarrow \mathcal{G}.get\_node(:r_{refd2}, id_{refd2})$
23. $\quad\quad e \leftarrow \mathcal{G}.add\_edge(v_{refd1}, v_{refd2}, :r, id)$
24. $\quad\quad$ **for all** $A_i \in R \,|A_i \notin pk(r) \wedge A_i \notin fks(r)$ **do**
25. $\quad\quad\quad e.add\_property(A_i, t[A_i])$
26. $\quad\quad$ **end for**
27. $\quad$ **end if**
28. **end for**

If a table is previously determined to be converted to edges, a new edge is created in the graph database for every tuple of the table. Nodes which the edge will connect are found using the labels (names of the referenced tables) and ids (concatenated values of attributes in each of the keys in a tuple). The new edge is labeled with the table name, and given the id which corresponds to the primary key value. At the end, all non-key attributes in a tuple are added as properties of the edge, in the same way it is done for the nodes.

Function $convert\_db$ (Algorithm 5) describes the main conversion process. After identifying the cyclic foreign keys and the migration order, tables are converted to nodes

and edges.

Algorithm 5 RDB to GDB Conversion: **convert_db**(**r**)

Input: relational database instance **r**
Output: graph database instance $\mathcal{G}$

```
1.  CyclicFKs ← find_all_cycle_ref(r)
2.  ToEdges, ToNodes ← get_migration_order(r)
3.  G ← ∅
4.  for all r ∈ ToNodes do
5.     table_to_graph(r, G)
6.  end for
7.  for all r ∈ ToEdges do
8.     table_to_graph(r, G)
9.  end for
10. for all fk ∈ CyclicFKs do
11.    r_refing ← refing(fk)
12.    r_refd ← refd(fk)
13.    for all t ∈ r_refing do
14.       id_refing ← t[pk(r)]
15.       id_refd ← t[fk]
16.       v_refing ← G.get_node(:r_refing, id_refing)
17.       v_refd ← G.get_node(:r_refd, id_refd)
18.       G.add_edge(v_refing, v_refd, :fk)
19.    end for
20. end for
```

Last part of the conversion process is creation of edges originated from cycle-forming foreign keys. This is the only part of the process where relational data is revisited, only to scan those tables with cycle-forming foreign keys, in order to determine which nodes should be connected by the new cycle-forming edge. Cycle-forming edges are created in the same way the other edges which originate from foreign keys are. The new edge is labeled with the foreign key name, and it has no additional properties.

6 EXPERIMENTS

During the testing period, we have conducted several conversions of the copies of OLTP databases used in different information systems, which were stored in IBM Informix relational database management system. Neo4j graph database was used as target database.

For illustration of our method's efficiency, we present here the comparison of the graph data model generated by our method with two models used by some other methods.

The data used in this experiment is part of a Stack Exchange [25] question and answer web sites aimed for various professionals (such as StackOverflow). We used a smaller set of data, specifically the site aimed towards database professionals (dba.stackexchange.com). Data consists of 8 relational database tables. For simplicity, we have omitted one of them (PostHistory), while others took part in the conversion processes (Posts, Users, Badges, Comments, PostLinks, Tags and Votes). In total, these tables contained 903.298 tuples, 57 attributes, with an average of 1,57 foreign keys per table.

Using this relational data as a source, conversion has been made into three different graph models:

- Zero-property model: Each tuple becomes a node, every attribute becomes a node connected to the tuple node, every foreign key becomes an edge. This model is used in approaches such as RDB2Graph.
- Node-property model: Each tuple becomes a node, every attribute becomes a node property, every foreign key becomes an edge. This model is used in approaches such as R2G.
- Full-property model: the model used in the method presented in this paper.

The graph description data for these models is shown in table 1. Last row shows the size of these databases implemented in Neo4j database system.

Table 1. Number of objects and size of target databases

model	zero-property	node-property	full-property
nodes	6.632.525	903.298	716.825
edges	4.631.988	1.092.647	928.240
props	0	2.206.509	2.206.509
size (MB)	6.473	3.275	3.392

Comparing to approaches mentioned in Section 2 which model all attributes as nodes (zero-property model), such as RDB2Graph, our proposition results in significantly smaller number of generated nodes and edges but with a lot of properties (all of them). Comparing to approaches which model all tuples as nodes with their attributes modeled as properties, such as R2G, our proposition results in about 20% smaller number of nodes, about 15% smaller number of edges and the same number of properties. Size of the databases in all three cases depends on the internal ways Neo4j stores data, so it is given here for illustrative purposes only.

Figure 1 shows the part of the imported data (posts related to user with id value 8) using the full-property model in the Neo4j graphical interface. Nodes with different labels have different colors. For graphical purposes, captions and sizes are set for each type of node and edge. On the bottom of the screen, the properties of the selected node (post with id 147) are visible.

Let us now explore the impact of these models to real-life queries upon a graph database. Stack Exchange already offers public access to queries regarding specific user

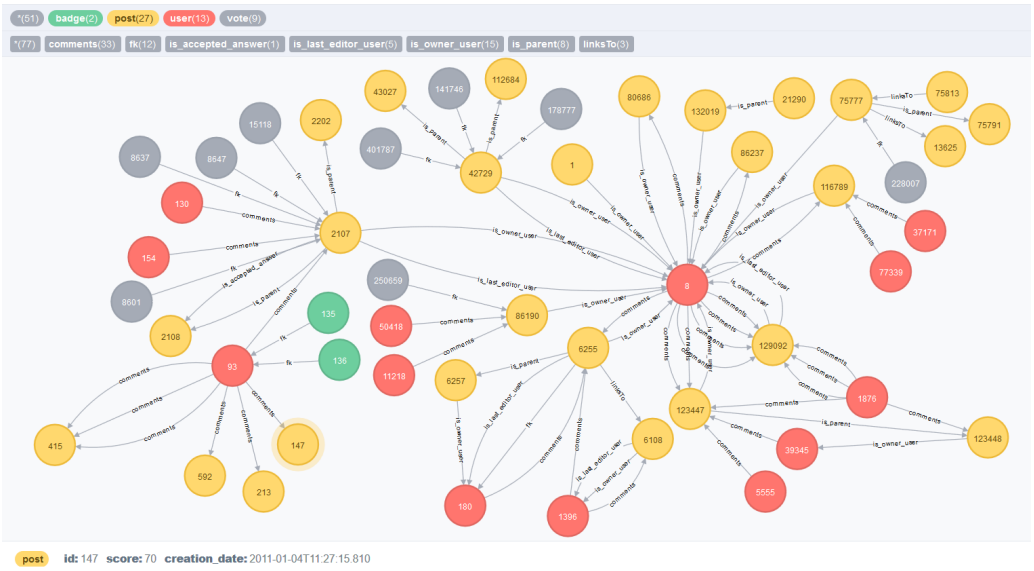


Fig. 1. Part of converted data shown in Neo4j browser

etc. [26]. We have built similar queries with a difference of addressing bigger amount of the data. These queries have been implemented in each model:

- q1: Users and their accepted answer ratio. Finds user names, total number of answers, number of accepted answers and accepted answer ratios, including only users with at least 10 answers and one accepted answer.
- q2: Users and their comment score. Finds user names, number of comments and average comment scores, including users with at least 20 comments.
- q3: Duplicated posts. Finds duplicated posts, including duplicates of duplicates, their duplicates, and so on, up to 4th level. (The limitation to the 4th level was introduced in order to get comparable results in reasonable time.).

During the experiment, the queries were implemented in Cypher programming language [27], executed on a server with a pre-cached data. The performance of the Neo4j database was measured. Each query executed 10 times, and the mean time is taken into account. Neo4j was running on a Linux Debian server, with a single Intel Xeon E7-4830 (2.13GHz) CPU and 7 Gb of memory assigned to the database engine.

Average times of performing each of the queries on all models are shown in figure 2. For clarity, times for each query are connected with a line. While the zero-property model shows the worst results for all queries, the results of the full-property model is slightly faster than

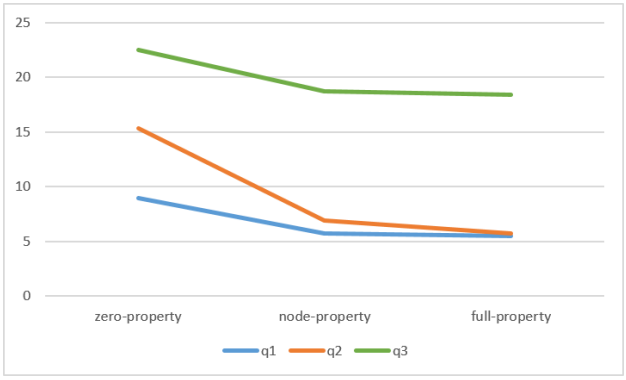


Fig. 2. Average times (s) of running queries q1, q2 and q3 on each model

the node-property model. Expectedly, average duration of each query corresponds to the model size - bigger models are slower than the smaller ones. It has to be noted though, that these queries examine many nodes (for example, in query q1 26% of nodes in the whole database are being examined). In most everyday cases, query will focus on a single node, its immediate neighbors and their properties, so these differences may not be noted. However, for graph mining purposes like finding frequent subgraphs and other queries which run through many graph elements, smaller models should be preferred.

7 FUTURE WORK

Implementation of the universal relational-to-graph database conversion process provides the means of apply-

ing various graph mining methods on the relational data. Our future work is twofold.

In the most part, analytics based on the relational data relies on the means of developing the analytic tools. For example, data warehouses can provide analytics based on available reports which in turn rely on the available data which came in through the ETL process. Both are designed and implemented by people who had an idea what should be analyzed. On the other hand, applying existing graph mining methods such as frequent subgraph mining to a converted graph database, may show connections between the data that were previously not considered, or even expected. A complex relations between tuples that belong to various tables might exists and reveal a new insight into that data. Based on patterns, irregularities could be found as well. Our first goal is to explore the usage of those graph mining methods which may help identify interesting complex and maybe unexpected data connections.

Over the years of RDBMS' domination, various types of data had been stored in relational databases, some of which is temporal and spatio-temporal data. Recently, the importance of dynamic graphs is emphasized. Static graph shows the data accumulated over time and possible overestimate relations between the data. Instead of that, dynamic graph (or temporal graphs) represent a graph that evolve over time, or time series of graphs. Given the increased research interest in temporal graphs analysis [28], our second goal is to explore the possibilities of creating a temporal graph from temporal or spatio-temporal relational database source.

8 CONCLUSION

The growing awareness of usefulness of big data analyses and graph mining algorithms, combined with the fact that the majority of human-produced data resides in relational databases, result in need of relational database to graph database conversion methods.

Our goal is to make the relational data available for graph mining algorithms, and in this paper, we have presented a method of converting the entire relational database to a graph database. The presented algorithms leverage the property graph data model, making use of both node's and edge's properties. Algorithms are independent of the data semantics, and provide the conversion without data loss. Since some relational databases allow cyclic relationships between tables, algorithms for identification such relationships are presented as well. Main idea behind the conversion algorithm is to read the relational data and load it into the graph database in as few steps as possible. This is accomplished by determining table's load order, and all the tables but those which are part of cyclic relationships are scanned only once. The tables taking part in cyclic relationships are revisited after the initial load, in order to establish new edges based on cycle-closing foreign keys.

Even though the entire database conversion can be time and space consuming process, we find it to be a practical solution in the case of performing repetitive multiple graph mining methods on the generated data which leverage features of the modern graph databases. Also, we find that the modeling dependent attributes as properties is the optimal way of maintaining all of the source data, given the total amount of relational data being converted.

Finally, we have verified the proposed process by applying it to the real world case and presented the results. We have compared the data models created by our method with those created by other methods and shown the advantages of full-property model regarding large-scale queries on a graph database.

REFERENCES

[1] N. Roussopoulos and J. Mylopoulos, "Using semantic networks for data base management," in *Proceedings of the 1st International Conference on Very Large Data Bases*, pp. 144–172, ACM, 1975.

[2] R. Soussi, *Querying and extracting heterogeneous graphs from structured data and unstrutured content*. PhD thesis, Ecole Centrale Paris, 2012.

[3] J. Park and S.-g. Lee, "Keyword search in relational databases," *Knowledge and Information Systems*, vol. 26, no. 2, pp. 175–193, 2011.

[4] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proceedings of the 31st international conference on Very large data bases*, pp. 505–516, VLDB Endowment, 2005.

[5] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 305–316, ACM, 2007.

[6] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: enabling keyword search over relational databases," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 627–627, ACM, 2002.

[7] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 670–681, VLDB Endowment, 2002.

[8] K. Xirogiannopoulos, U. Khurana, and A. Deshpande, "Graphgen: Exploring interesting graphs in relational data," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 2015.

[9] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao, "Large-scale graph analytics in aster 6: bringing context to big data discovery," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1405–1416, 2014.

[10] S. Palod, "Transformation of relational database domain into graphs based domain for graph based data mining," *Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, SAD*, 2004.

[11] J. F. Sowa, "Conceptual graph summary."

[12] S. Pradhan, S. Chakravarthy, and A. Telang, "Modeling relational data as graphs for mining.," in *COMAD*, Citeseer, 2009.

[13] R. De Virgilio, A. Maccioni, and R. Torlone, "Converting relational to graph databases," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 1, ACM, 2013.

[14] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker, "Vertexica: your relational friend for graph analytics!," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1669–1672, 2014.

[15] J. Fan, A. Gerald, S. Raj, and J. M. Patel, "The case against specialized graph analytics engines,"

[16] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[17] "Allegrograph database." `http://www.franz.com/agraph/allegrograph`. Accessed: 2016-09-21.

[18] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey, "Dex: high-performance exploration on large graphs for information retrieval," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pp. 573–582, ACM, 2007.

[19] "Dex graph database." `http://sparsity-technologies.com/{#}sparksee`. Accessed: 2016-09-21.

[20] B. Iordanov, "Hypergraphdb: a generalized graph database," in *Web-Age information management*, pp. 25–36, Springer, 2010.

[21] "Hypergraphdb." `http://www.hypergraphdb.org`. Accessed: 2016-09-21.

[22] "Neo4j graph database." `http://www.neo4j.com`. Accessed: 2016-09-21.

[23] "Objectivity infinite graph." `http://www.objectivity.com/products/infinitegraph/`. Accessed: 2016-09-21.

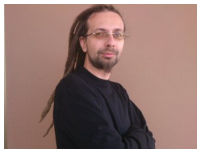
[24] R. Angles, "A comparison of current graph database models," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 171–177, IEEE, 2012.

[25] "Stack exchange." `http://stackexchange.com/`. Accessed: 2016-08-03.

[26] "Stack exchange data explorer." `http://data.stackexchange.com/`. Accessed: 2016-08-03.

[27] "Cypher query language." `http://neo4j.com/developer/cypher-query-language/`. Accessed: 2016-09-21.

[28] P. Holme, "Modern temporal network theory: a colloquium," *The European Physical Journal B*, vol. 88, no. 9, pp. 1–30, 2015.



Ognjen Orel is the head of Academic Information Systems department at the University of Zagreb, University Computing Centre. His research interests include databases, information systems, graphs, data and graph mining and analyses. Currently, he is the leader of the Croatian Qualifications Framework Registry Information System project.

Slaven Zakošek is an assistant professor in Computer Science at the University of Zagreb, Faculty of Electrical Engineering and Computing. His research interests include databases, database systems, information systems, software architecture and software engineering. Currently, he is the leader of the project ISOHOPS – Information system for maintenance management in Croatian transmission system operator.

Mirta Baranović is a full professor in Computer Science at the University of Zagreb, Faculty of Electrical Engineering and Computing. Her research interests include databases, information systems, data warehouses, data lakes, and semantic web. She served as Vice Dean for students and Education at the University of Zagreb, Faculty of Electrical Engineering and Computing. Currently, she is a member of the Croatian Centre of Research Excellence for Data Science.

AUTHORS' ADDRESSES

Ognjen Orel, M.Sc.
University Computing Centre SRCE,
University of Zagreb,
J. Marohnića 5, HR-10000, Zagreb, Croatia
email: ognjen.orel@srce.hr

Slaven Zakošek, Ph.D.
Prof. Mirta Baranović, Ph.D.
Department of Applied Computing,
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000, Zagreb, Croatia
email: {slaven.zakosek, mirta.baranovic}@fer.hr

Received: 2015-11-29
Accepted: 2016-12-23